Butler University LIBRARIES

Butler Journal of Undergraduate Research

Volume 3 Article 11

2017

Tango: A Spanish-Based Programming Language

Ashley M. Zegiestowsky
Butler University, ashleyzeg@gmail.com

Tango: A Spanish-Based Programming Language

Cover Page Footnote
Thank you to Ankur Gupta and Alex Quintanilla for feedback on an earlier version of this thesis.

TANGO: A SPANISH-BASED PROGRAMMING LANGUAGE

ASHLEY M. ZEGIESTOWSKY, BUTLER UNIVERSITY
MENTOR: JONATHAN SORENSON

Abstract

The first part of this article deals with the creation of my own Spanish-based programming language, Tango, using Spanish key words (instead of English key words). The second part relates to the design and implementation of a compiler that follows the grammar rules outlined in the Tango language in order to successfully lexically analyze, parse, semantically analyze, and generate code for Tango. This article begins with a description of the specific goals achieved in the Tango language, an explanation and brief examples of the Tango Grammar, a high-level overview of the compiler design and data structures used, and concludes with ideas for future work and helpful advice. The full grammar, list of keywords, and source code for the compiler can be found in the Appendices.

Introduction

This project, to write my own Spanish programming language, provided me with a way to combine both of my fields of study, Computer Science and Spanish. Before starting this project, I knew very little about the details that went into designing a programming language and compiler. From a high-level perspective, I learned about the importance of context-free grammars and the different phases that make up a compiler, which will be explained in further detail later.

Specific Goals

In creating the Tango programming language as well as the accompanying compiler, there are five main goals around which I centered this article. These goals are listed as follows:

1. To create a user-friendly programming language utilizing key words and a syntactic structure that closely resembles the Spanish language.

2. To design the language with a similar technical feel to that of some of the most widely used languages in both an educational and professional environment, such as Java and C++.
3. To allow the compiler to read and interpret both keywords and variable identifiers with special characters native to the Spanish language, such as accents and tildes above certain letters.
4. To provide a compiler that cross-compiles to run against the Java compiler instead of directly generating machine code.
5. To assure that the Tango language is Turing complete, or computationally universal.

The first goal forms an essential backbone to the entire project as a whole. There are over thousands of different programming languages utilized and available across a wide array of countries and cultures, and new languages and frameworks are being created everyday. However, a huge majority of these programming languages are English-based, meaning the keywords and syntactic structure are based on the English language [6]. Less than a handful of Spanish-based programming languages have been created including (but not limited to):

- GarGar, a Spanish procedural programming language based on Pascal for learning purposes [6].
- Latino, a language with completely Spanish-based syntax [6].
- RoboMind, an educational programming language available in multiple languages (including Spanish) [6].

Although the above mentioned Spanish-based programming languages accomplish a similar goal to the Tango language, Tango differs from these pre-existing languages in three key areas. First, the Tango language and compiler incorporate and support the usage of special characters (such as the accent mark and tilde) in both the key words of the language as well as any variables, constants, and function names generated by the user. Second, the compiler implementation utilizes cross-compilation to Java which facilitates simple installation and usage. The only requirement to successfully to download, compile, and execute a program written using Tango is to have a stable version of Java installed, while the Latino language, for example, lists thirteen different machine requirements in its installation documentation. Third, the key words and syntactical structure of Tango have a deliberate "Java-like" flavor with the intent to ease the transition to the usage of Java (or any similar language) given that Java is one of the most commonly used

server-side languages in the workforce today. All three of these differences either improve upon the existing implementations of Spanish based programming languages or offer additional features not present in the languages mentioned earlier that are more analogous to the Spanish language itself.

The purpose and goal behind creating Tango is to contribute to the growing technical community that spans across many languages and cultures. With the creation of this prototype version of a Spanish-based programming language, the educational and professional reach of such a tool could prove to be very beneficial in engaging and inspiring both young minds and experienced professionals within the field of technological development.

The second of the above-mentioned goals relates closely to the first goal in the sense of the basic design and feel of the programming language. It is essential that Tango is both intuitive to a native Spanish speaker, yet also compatible from a logical and syntactical standpoint to other more commonly used programming languages.

The third goal, which deals with the handling of special characters, proves to be highly unique to Tango. Unlike the English language, the Spanish language employs the use of accents and tildes above select letters forming characters that do not exist in English. These accents and/or tildes are essential to the language and are crucial to the meaning of certain words. The lack of an accent or tilde could change the meaning of the word entirely. Therefore, allowing for the compiler to adequately scan and process the possibility of these special characters is vital to the correctness and representation of the Spanish language as demonstrated in the Tango programming language.

The fourth goal mentioned pertains to a more technical implementation of compiler design. When designing the compiler, I chose to cross-compile to Java instead of generating machine code directly in the code generation phase. This decision, given the relatively short time provided to complete such an ambitious project, allowed me to create a more verbose and comparatively functioning programming language. If I had chosen to implement a compiler that generated machine code, the final product would be more limited both in scope and overall functionality. Another benefit to choosing to cross-compile to Java is that the Java compiler is machine independent. Also important to note, the target language is Java and the compiler is written in Java. However, these two choices are independent of

one another. More specifics of the compiler design and implementation will be discussed later.

The fifth and final goal assures that the Tango language will be Turing complete, meaning computationally universal. A language is considered Turing complete if it can be used to simulate any single-taped Turing machine [9]. Moreover, a "Turing machine can do everything that a real computer can do" [9]. A universal language does not have to be complex [2]. In the case of Tango, the language itself is not extremely complex, but it does meet the requirements to be considered Turing complete.

Overall, these five main goals helped drive and inspire both the research and implementation process necessary to successfully create a prototype programming language based primarily on the Spanish language.

Context-Free Grammar (CFG)

CONTEXT-FREE GRAMMAR AND TANGO

The grammar for the Tango programming language is a combination of original productions and recognized production patterns from outside resources. The basic definition of a context-free grammar is as follows:

> A grammar consists of a collection of substitution rules, also called productions. Each rule appears as a line in the grammar, comprising a symbol and a string separated by an arrow. The symbol is called a variable, or nonterminal. The string consists of variables and other symbols called terminals. One variable is designated as the start variable [9].

The purpose of a context-free grammar is to provide rules from which a "syntactically valid string of terminals" can be generated [7]. The process of generating a valid string of terminals is referred to as a derivation that can be described as "a series of replacement operations that shows how to derive a string of terminals from the start symbol" [7]. This same information can be represented pictorially with the use of a parse tree (which will be utilized later when providing sample productions) [9].

There are two main types of derivations: rightmost derivations and leftmost derivations. For the purposes of simplicity and consistency, the select examples shown will be utilizing a leftmost derivation meaning that "in every step of the derivation, the leftmost nonterminal" is selected for replacement [5]. Furthermore, the Tango grammar can be classified as LL(1)

which stands for the leftmost derivation when the input is scanned from left to right with one-token look ahead. To further clarify, a grammar is said to be LL(1) given that, "any two rules defining the same nonterminal must have disjoint selection sets [1]. Meeting the condition to declare a grammar LL(1) is crucial in order to construct a recursive descent parser which will be further explained in Compiler Design.

SPANISH LANGUAGE INFLUENCE & NUANCES IN GRAMMAR

Other important aspects of the Tango grammar worth mentioning relate to the selection of key words and syntactic structures in regards to the Spanish language. Three main challenges occurred when selecting key words and syntax:

1. How should the unpredictability of masculine or feminine nouns or descriptive keywords be handled?
2. What verb tense should be used when using verb-like keywords?
3. How should the overall syntax be structured in a way that closely resembles the overall structure of the Spanish language?

As for question one, a simple solution was utilized in order to account for descriptive keywords with different character endings depending upon the object being described. For example, in the Spanish language masculine words end in the letter 'o' where as feminine words in the letter 'a'. Since these word endings are dependent upon the noun or object being described, this created a problem of consistency when selecting keywords. However, instead of such words ending in an 'o' or 'a', all key words that fit this criteria end with the symbol '@'. The symbol '@' has become more commonly used and accepted within the Spanish community for words that could be either masculine or feminine. Some of the keywords for the Tango language that fall under this category include: nul@ (null), vaci@ (void), públic@ (public), nuev@ (new), ciert@ (true), fals@ (false). Note, the word in parenthesis following the Spanish key word is an English equivalent for reference for those unfamiliar with the Spanish language.

As for the second question regarding verb tense for verb-like keywords, yet another simple solution was implemented. In most English-based programming languages examples of verb-like keywords include words such as do, return, and print, to list a few [10]. In English, these verbs are in a command form (which there is only one kind of command conjugation for each verb regardless of the subject or audience). However, in Spanish, there are

multiple different ways to conjugate the command form of a verb and even more differences amongst different countries and dialects. In order to establish a consistent representation of verb-like keywords that would be understood by all Spanish speakers, the infinitive form of the verb serves as the best method of representation. For example, the following are Spanish verb-like keywords present in the Tango programming language: hacer (do), regresar (return), imprimir (print). Again, the word in parenthesis, is the English equivalent to the Spanish key word.

As for the third and final question, mirroring the Spanish language from a syntactical perspective proved to be the most difficult to emulate. However, one specific way in which this is prevalent in the grammar is the placement of the access identifiers in relation to a function definition. In the English language, adjectives are typically placed before the noun they are describing; take for example the phrase, "the long red dress". In the Spanish language, adjectives are typically placed after the noun they are describing: take for example, the phrase, "el vestido largo y rojo" (which would directly translate to English as "the dress long and red"). This same principle can be seen in the syntax relating to a function definition demonstrated in **Figure 1**.

Example Java Function Definition (English):

```
public void func_name (int param1, int param2) {…}
```

Example Tango Function Definition (Spanish):

```
func func_name públic@ vaci@ (ent param1, ent
param2) {…}
```

Figure 1. Note the differences between the two function definitions are very subtle but reflect the structural nuances of the Spanish language.

SAMPLE TANGO PRODUCTIONS AND DERIVATIONS

This section will walk through a simple example of productions and derivations using Tango's grammar. The full grammar and list of keywords (as well as their relative English equivalent) are listed in detail in **Appendix A**.

As mentioned at the beginning of this section, Tango's grammar is a mix of original productions and predefined productions from outside sources. Henceforth, the Tango grammar utilizes portions of the calculator grammar

detailed in Michael Lee Scott's Programming Language Pragmatics [7]. Select rules, or productions, are outlined and displayed in **Figure 2**:

```
stmt → id = expr;
expr → term term_tail
term_tail → add_op term term_tail |
term → factor factor_tail
factor_tail → mult_op factor factor_tail |
factor → ( expr ) | id | number
add_op → + | -
mult_op → * | /
```

Figure 2. Grammar productions (rules) from the LL(1) calculator grammar [7].

As mentioned earlier, only a small portion of the Tango grammar is illustrated in the above figures (Figure 2-6). To see the complete grammar and list of keywords, refer to **Appendix A**.

Compiler Design

The compiler design and implementation comprise a large portion of this article and are closely related to the grammar outlined in the previous section. The compiler construction can be broken into four different phases: lexical analyzer (scanner), parser, semantic analyzer, and code generator [2].

The lexical analyzer, or scanner, scans the source program character by character "recognizing which strings of symbols from the source program represent a single entity, or token" [2]. The lexical analyzer also identifies and categorizes the tokens according to their value in relation to the rest of the program. Some of the token types present in the Tango scanner include: keywords, variable identifiers, numeric values, arithmetic operators, special characters, etc.

FIRST

stmt{ id }

expr{ (, id, number }

term_tail { +, -}

term { (, id, number }

factor_tail { *, / }

factor { (, id, number }

add_op { +, - }

mult_op { *, / }

FOLLOW

id { +, -, *, /, =, id}

number { +, -, *, /,), id }

({ (, id, number }

) { +, - , *, /, id}

= { (, id, number }

+ { (, id, number }

- { (, id, number }

* { (, id, number }

/ { (, id, number }

expr {), id, ; }

term_tail {), id }

term { +, -,), id }

factor_tail { +, -,), id }

factor { +, -, *, /,), id }

add_op { (, id, number }

mult_op { (, id, number }

PREDICT

1. stmt–→id = expr {id}
2. expr –→ term term_tail {(, id, number}
3. term tail –→ add_op term term_tail {+, -}
4. term tail –→ {), id}
5. term –→ factor factor_tail {(, id, number}
6. factor tail –→ mult_op factor factor_tail {*, /}
7. factor tail –→ {+, -,), id}
8. factor–→ (expr) {(}
9. factor–→ id {id}
10. factor –→ number {number}
11. add_op–→ + {+}
12. add_op–→ - {-}
13. mult_op–→ * {*}
14. mul_top–→ / {/}

Figure 3. First, Follow, & Predict Sets for the grammar productions in **Figure 2** [7].

```
stmt → id = expr;
stmt → x = expr;
stmt → x = term term_tail;
stmt → x = factor factor_tail term_tail;
stmt → x = 5 factor_tail term_tail;
stmt → x = 5 term_tail;     (* factor_tail
→ )
stmt → x = 5;     (* term_tail → )
```

Figure 4. Leftmost derivation for the string, `x=5;`

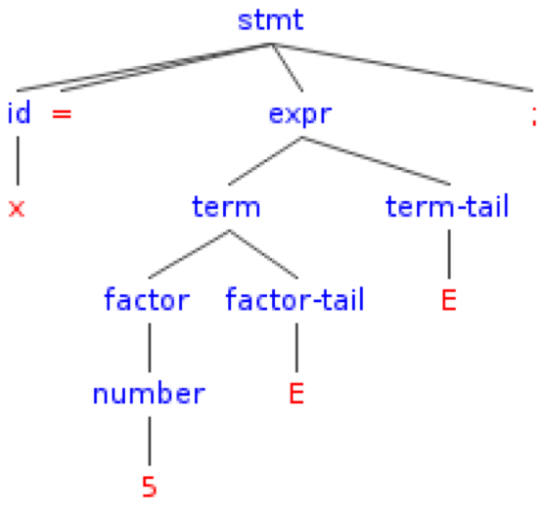

Figure 5. Parse Tree for the string, `x=5;` [8]. Note: E stands for ε, the empty string

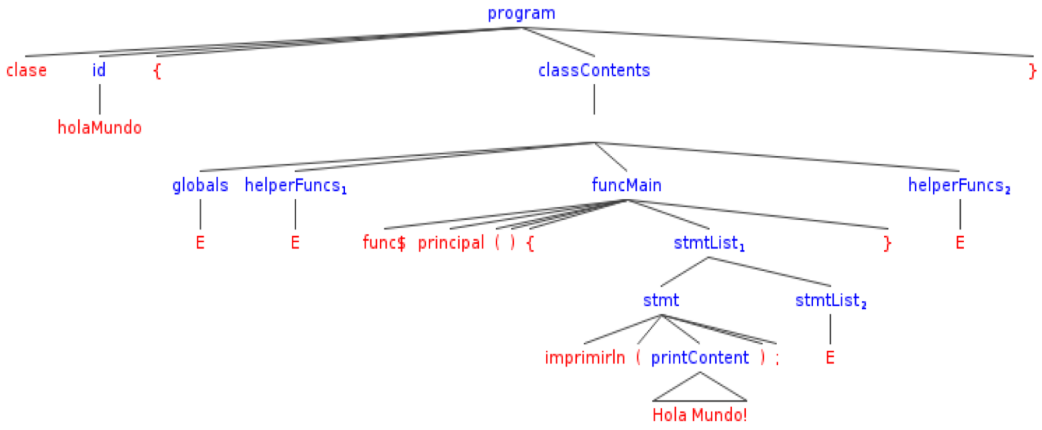

Figure 6. Parse tree for the sample Hello World program as seen in **Appendix B** [8]. Note: E stands for ε, the empty string

The second phase is the parser. Since the Tango language has an LL(1) context-free grammar, a recursive descent parser is the method utilized in completing this phase of the compiler. The parser will check for "proper syntax, issue appropriate error messages, and determine the underlying structure of the source program" [1]. The recursive descent parser illustrates "top-down (predictive) parsing" which relates directly to the grammar [7].

The third phase consists of semantic analysis. The semantic analyzer is intertwined with the parser. Whereas the parser checks the program for syntactic correctness according to the grammar rules, the semantic analyzer checks data types and other logical checks that can not be performed by the parser alone [1].

The fourth and final phase is code generation. During the code generation phase, a traditional compiler translates the successfully parsed tokens or syntax trees into "machine language (binary) instructions, or to assembly language" [2]. As mentioned previously in the section on Specific Goals, the Tango compiler does not directly generate machine code. In fact, the tango compiler, in the code generation phase, generates equivalent Java code, which is then run against the Java compiler. The reasoning for choosing to cross compile to Java is explained in further detail in the section on Specific Goals.

If an error occurs during any phase of the compiler (lexical scanner, parser, semantic analyzer, or code generator), the compiler terminates and an

error message is displayed in the console with the line number and a potential error message depending on the error.

CODE STRUCTURE & DATA STRUCTURES

The overall structure of the compiler is a Java based application with five classes:

- *TangoCompiler.java* – This is the main class from which the program runs. When the program starts, the program prompts the user to enter the name of an input file to be processed. Instances of the *TangoScanner* and *TangoParser* objects are instantiated in order to begin processing the source program.
- *TangoScanner.java* – This class constitutes the lexical analysis, or scanner, phase of the compiler. The *TangoScanner* class utilizes instances of the *Token* class in order to create an array of tokens. Each time the scanner recognizes a new Token, a new Token object is created and added to the array of tokens which utilizes the ArrayList data structure.
- *Token.java* – This class contains all of the details and data related to the different token types available (including keywords) in the Tango language. A mix of simple arrays and single line functions are utilized inside the *Token* class. The array of Token objects is then processed in the *TangoParser* in which some of the simple functions created in the Token are used to perform necessary checks when moving through the parsing process. The *Token* class is also where the symbol table is stored which uses the Hash Table data structure.
- *TangoParser.java* – This class is the most intensive portion of the compiler. The *TangoParser* parses the array of tokens produced by the scanner using a recursive descent parser. Therefore, recursion is employed in order to move through the token stream. Semantic analysis occurs during this phase in which stack data structures are utilized. The *CodeGenerator* class is also instantiated inside of the parser and recursively generates Java code as the tokens are parsed.
- *CodeGenerator.java* – This class uses a FileWriter in order to write the equivalent Java code to a new file. The class consists of a multitude of void functions that are called within the recursive descent parser in order to successfully write to a file the newly generated code that can

then be compiled and run against the java compiler using the javac and java command.

Conclusion

FUTURE WORK

After investing close to eight months on the creation of the Tango programming language and compiler, I am proud to say that I have produced a functional, yet limited, prototype language. If I had more time to invest in this project, there is plenty of work left in order to make Tango a fully functional and bug free programming language that could be utilized primarily for educational purposes. Some of the additions and improvements, I would integrate into the language would include:

- *Expand the Functionality* – With such limited time, I was only able to implement some of the most basic functionality. Integrating more complex data structures, object-oriented principles, and additional libraries would both complement and improve upon the existing source code.
- *Optimization & Debugging* – Very little thought or effort was put into optimizing both performance and memory when implementing the compiler. This would be an important improvement in order to truly test the power and limitations of the compiler.
- *Modify Code Generation* – The current complier cross compiles to Java by generating equivalent Java code that is run against the Java compiler. By modifying the code generation phase to directly generate machine code instead of Java code would allow the Tango language to no longer be dependent upon the Java compiler. Although this may increase efficiency and performance for the Tango compiler, this implementation would limit Tango to a fixed platform.
- *Interactive Website* – It would be beneficial as well to create an online platform or downloadable resource to which the public could access freely and directly in order to both write and run Tango programs. Along with this resource, creating a simple tutorial to aid users in programming in the Tango language with a link to the complete documentation would greatly complement all of the work that has been put forth in creating this project.

HELPFUL ADVICE

To anyone interested in completing a similar project, there are several tidbits of advice that are helpful and important to keep in mind when undergoing a project of this magnitude. First of all, make a timeline of important goals and milestones, and stick to it! Although this is a simple and obvious strategy when undergoing any type of project, it becomes even more crucial when dealing with a large code base with lots of moving parts.

Secondly, the grammar, derivations, and syntax trees are just as important as the actual code written to implement the compiler. If your grammar has errors (such as not meeting LL(1) criteria or highly ambiguous), then your compiler will have errors (making it even more difficult to implement). It is easier to fix an error when it is still just a grammar rule rather than when it has already been faultily integrated into the compiler. Lastly, start small and build up from there. It is tempting to want to implement everything at once. However, start with the basic functionality and then continue to modify and expand the language and compiler accordingly. Those are some of the pieces of advice that I would give to anyone with the desire to delve into compiler theory and design.

Whether or not the Tango programming language will ever be used or viewed outside the scope of this undergraduate thesis does not equate to the success or failure of the final product produced. The learning curve, work ethic, and technical knowledge I gained from completing this project cannot be monetized. Nevertheless, I have attained a new-found pride in the progress I have made in regards to the Tango language and developed a well-earned confidence in my technical abilities as a whole. The Tango programming language demonstrates a small but vital attempt to widen the reach of technological advancement across language and cultural barriers beginning in the educational realm and edging towards a professional environment. The language itself is young and in need of maturation and finesse. If nothing else, Tango can serve as a jumping off point and inspiration into the world of Spanish-based programming languages.

References

Bergmann, Seth. *Compiler Design: Theory, Tools, and Examples*. Dubuque, Iowa: W.C. Brown Publishers, c, 1994.

Brookshear, J. Glenn, David T. Smith, and Dennis Brylow. *Computer Science: An Overview*. 11th ed. Harlow: Addison-Wesley, 1997.

Galimberti Jarman, Beatriz, et al. *The Oxford Spanish Dictionary: Spanish-English/English-Spanish*. New York: Oxford UP, 2003.

"Online Language Dictionaries." English-Spanish Dictionary. http://www.wordreference.com/es/translation.asp

Parsons, Thomas W. *Introduction to Compiler Construction*. New York: Computer Science Press, 1992.

Pigott, Diarmuid. *HOPL: An interactive roster of programming languages*. Murdoch University, School of Information Technology, 1995.

Scott, Michael Lee. *Programming Language Pragmatics*. 3rd ed. Amsterdam: Elsevier/Morgan Kaufmann Pub., 2009.

Shang, Miles. Syntax Tree Generator. 2011. Web. http://mshang.ca/syntree/

Sipser, Michael. *Introduction to the Theory of Computation*. 3rd ed. Boston, MA: Cengage Learning, 2013.

Weiss, Mark Allen. *Data Structures & Problem Solving using Java*. Boston: Pearson/Addison Wesley, 2010.

Appendix A: List of Keywords & Full Grammar

ent	int
dec	double
cadena	String
lista	[]
bool	Boolean
ciert@	true
fals@	false
nuev@	new
si, sino, si, sino	if, else, if, else
para	for
mientras, hacer mientras	while, do while
clase	class
func$ principal()	public static void main(String [] args)
estátic@	static
vaci@	void
públic@	public
nul@	null
regresar	return
escáner	scanner
imprimirln	println
imprimir	print
sigEnt	nextInt
# (single line comment)	// (single line comment)

*Both wordrefernce.com [4] & The Oxford Spanish Dictionary [3] were referenced when selecting the appropriate Spanish keyword.

FULL GRAMMAR

High Level Productions

program → *clase* id accessMod { classContents }

accessMod → *públic@*

classContents → funcMain

funcMain → *func$ principal*() { stmtList }

stmtList → stmt stmtList |

Library Call Productions

stmt → *imprimirln*(printContent);

printContent → “stringValue” | id

Declaration & Assignment Productions

stmt → id = expr;

stmt → dataType id decTail;

decTail → = expr |

dataType → *ent*

dataType → *dec*

dataType → *cadena*

dataType → *bool*

expr → term termTail

expr → boolOp

termTail → addOp term termTail |

term → factor factorTail

factorTail → multOp factor factorTail |

factor → (expr)

factor → id

factor → number

addOp → +

addOp → -

multOp → *

multOp → /

boolOp → *ciert@*

boolOp → *fals@*

If Statement Productions

stmt → *si* (condition) { stmtList } siTail

siTail → *sino* sinoTail |

sinoTail → { stmtList } | *si* (condition) { stmtList } siTail

condition → expr conditionTail

conditionTail → compOp expr | ε

compOp → == | != | > | < | >= | <=

While Loop Productions

stmt → *mientras* (condition) { stmtList }

stmt → *hacer* { stmtList } *mientras* (condition);

FIRST, FOLLOW, PREDICT SETS[1]

Key

Red = keyword/terminal

Blue = terminal

Normal = non-terminal

High Level Sets

First

program { clase }

accessMod { públic@ }

classContents { func$ }

funcMain { func$ }

stmtList { mientras, hacer, si, id, ent, dec, cadena, bool, imprimirln }

stmt { mientras, hacer, si, id, ent, dec, cadena, bool, imprimirln }

Follow

classContents { '}' }

funcMain { '}' }

stmtList { '}' }

stmt { '}', mientras, hacer, si, id, ent, dec, cadena, bool, imprimirln }

Predict

program → clase id accessMod { classContents } { clase }

accessMod → públic@ { públic@ }

classContents → funcMain { func$ }

funcMain → func$ principal() { stmtList } { func$ }

[1] Separated by similar grammar concepts for organization

stmtList → stmt stmtList { mientras, hacer, si, id, ent, dec, cadena, bool, imprimirln}

stmtList → { '}' }

Library Call Sets

First

stmt { imprimirln }[2]

printContent { ", id }

Follow

printContent { ')' }

Predict

stmt → imprimirln(printContent); { imprimirln }

printContent → " stringValue " { " }

printContent → id { id }

Declaration & Assignment Sets

First

stmt {id, ent, dec, cadena, bool}[3]

dataType { ent, dec, cadena, bool }

decTail { =, }

expr { '(', id, number, ciert@, fals@ }

term { '(', id, number}

termTail { +, -, }

factor { '(', id, number }

factorTail { *, /, }

---

[2] Limited to FIRST set of library call productions

[3] Limited to FIRST set of declaration productions

addOp { +, - }

multOp { *, / }

Follow

id { =, *, /, +, -, ;, ')' }

number { =, *, /, +, -, ;, ')' }

= { '(', id, number, ciert@, fals@ }

({ '(', id, number, ciert@, fals@ }

) { *, /, +, -, ;, ')' }

+ { '(', id, number }

- { '(', id, number }

* { '(', id, number }

/ { '(', id, number }

ciert@ { ;, ')' }

fals@ { ;, ')' }

ent { id }

dec { id }

cadena { id }

bool { id }

stmt { '}', mientras, hacer, si, id, ent, dec, cadena, bool, imprimirln}

dataType { id }

decTail { ; }

expr { ;, ')' }

term { +, - , ;, ')' }

termTail { ;, ')' }

factor { *, /, + , -, ;, ')' }

factorTail { +, -, ;, ')' }

addOp { '(', id, number}

multOp { '(', id, number}

Predict

stmt → id = expr; {id}

stmt → dataType id decTail; { ent, dec, cadena, bool }

dataType → ent { ent }

dataType → dec { dec }

dataType → cadena { cadena }

dataType → bool { bool }

decTail → = expr { = }

decTail → { ; }

expr → term termTail { '(', id, number }

expr → boolOp { ciert@, fals@ }

termTail → addOp term termTail { +, - }

termTail → { ;, ')' }

term → factor factorTail { '(', id, number }

factorTail → multOp factor factorTail { *, / }

factorTail → { +, -, ;, ')' }

factor → (expr) { (}

factor → id { id }

factor → number { number }

addOp → + { + }

addOp → - { - }

multOp → * { * }

multOp → / { / }

boolOp → ciert@ { ciert@ }

boolOp → fals@ { fals@ }

If Statement Sets

First

stmt { si }[4]

siTail { sino }

sinoTail { '{', si }

condition { '(', id, number, ciert@, fals@ }

conditionTail { == , != , > , < , >= , <= }

compOp { == , != , > , < , >= , <= }

Follow

stmt { '}', mientras, hacer, si, id, ent, dec, cadena, bool, imprimirln }

siTail { si, '(', id, number, ciert@, fals@, imprimln, '}' }

sinoTail { si, '(', id, number, ciert@, fals@, imprimln, '}' }

condition { ')' }

conditionTail { ')' }

compOp { '(', id, number, ciert@, fals@ }

Predict

stmt → si (condition) {stmtList} siTail { si }

siTail → sino sinoTail { sino }

siTail → { si, '(', id, number, ciert@, fals@, imprimln, '}' }

sinoTail → {stmtList} { '{' }

sinoTail → si (condition) {stmtList} siTail { si }

condition → expr conditionTail { '(', id, number, ciert@, fals@ }

conditionTail → compOp expr { == , != , > , < , >= , <= }

conditionTail → { ')' }

compOp → == { == }

compOp → != { != }

[4] Limited to FIRST set of if statement productions

compOp → < { < }

compOp → > { > }

compOp → <= {<=}

compOp → >= {>=}

While Sets

First

stmt { mientras, hacer }[5]

Follow

stmt { '}', mientras, hacer, si, id, ent, dec, cadena, bool, imprimirln}

Predict

stmt → mientras (condition) {stmtList} { mientras }

stmt → hacer {stmtList} mientras (condition); { hacer }

[5] Limited to FIRST set of while loop productions

Appendix B: Example Programs

EXAMPLE PROGRAM #1: HOLA MUNDO! (HELLO WORLD!)

Tango Sample Code

```
clase holaMundo públic@ {
    func$ principal() {
        #variable declaration
        bool URC = ciert@;

        #if statement
        si ( URC ) {
imprimirln(“Hola Mundo! Hoy es el URC”);
        }
sino {
imprimirln(“Hola Mundo! Hoy NO es el URC”);
        }
    }
}
```

Generated Java Code

```
public class holaMundo {
     public static void main(String [] args ) {
          #variable declaration
          Boolean URC = true;

          #if statement
          if ( URC ) {
               System.out.println(“Hello  World!  Today  is
               the URC”);
          } else {
               System.out.println(“Hello  World!  Today  is
               NOT the URC”);
          }
     }
}
```

EXAMPLE PROGRAM #2: JUEGO DE ADVINIVAR (GUESSING GAME)

Tango Sample Code

```
clase juegoDeAdivinar públic@{
      func$ principal() {
            ent n = 27;  #hard coded number for now
            ent usuario = 0;
            escáner e = escáner() nuev@;

            mientras (usuario != n) {
                  imprimirln("Elige un número entre 1 y
                  100");
                  usuario = e.sigEnt();

                  si (usuario < 1) {
                        imprimirln("Su número es invalido.");
                        imprimirln("Elige un número entre 1 y
                        100");
                  } sino si(usuario > 100) {
                        imprimirln("Su número es invalid.");
                        imprimirln("Elige un número entre 1 y
                        100");
                  } sino si(usuario > n) {
                        imprimirln("Que     boludo...demasiado
                        alto!");
                  } sino si(usuario < n) {
                        imprimirln("Que     idiota...demasiado
                        bajo!");
                  } sino {  #usuario == n
                        imprimirln("Perfecto!            Está
                        correcto!");
                  }
            }
            imprimirln("Gracias por jugar!");
      }
}
```

Generated Code

```
import java.util.Scanner;

public class juegoDeAdvinar {
      public static void main(String [] args ) {
            int n = 27; //hard coded number for now
            int user = 0;
            Scanner e = new Scanner(System.in);

            while (user != n) {
                  System.out.println(“Choose a number b/w 1 &
                  100”);
                  user = e.nextInt();
                  if(user < 1) {
                        System.out.println(“Your   number   is
                        invalid”);
                        System.out.println(“Choose  number  b/w
                        1 & 100”);
                  } else if (user > 100) {
                        System.out.println(“Your   number   is
                        invalid”);
                        System.out.println(“Choose  number  b/w
                        1 & 100”);
                  } else if (user > n) {
                        System.out.println(“Too High!”);
                  } else if (user > n) {
                        System.out.println(“Too Low!”);
                  } else { //user == n
                        System.out.println(“Perfect!  You  got
                        it right!”);
                  }
            }
            System.out.println(“Thanks for playing!”);
      }
}
```

Appendix C: Source Code

See the full source code repository via the online resource GitHub: https://github.com/ashleyzeg/HonorsThesis.